12-2012

Symbolic model-checking of stateful timed CSP using BDD and digitization

Truong Khanh NGUYEN

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Jin Song DONG

Citation

NGUYEN, Truong Khanh; SUN, Jun; LIU, Yang; and DONG, Jin Song. Symbolic model-checking of stateful timed CSP using BDD and digitization. (2012). *Proceedings of the 14th International Conference on Formal Engineering Methods, , ICFEM 2012, Kyoto, Japan, November 12-16*. 398-413.
Available at: https://ink.library.smu.edu.sg/sis_research/5024

Symbolic Model-Checking of Stateful Timed CSP Using BDD and Digitization*

Truong Khanh Nguyen[2], Jun Sun[1], Yang Liu[3], and Jin Song Dong[2]

[1] ISTD, Singapore University of Technology and Design
sunjun@sutd.edu.sg
[2] School of Computing, National University of Singapore
{truongkhanh,dongjs}@comp.nus.edu.sg
[3] Temasek Lab, National University of Singapore
tslliuya@nus.edu.sg

Abstract. Stateful Timed CSP has been recently proposed to model (and verify) hierarchical real-time systems. It is an expressive modeling language which combines data structure/operations, complicated control flows (modeled using compositional process operators adopted from Timed CSP), and real-time requirements like $deadline$ and $within$. It has been shown that Stateful Timed CSP is equivalent to closed timed automata with silent transitions, which implies that the timing constraints of Stateful Timed CSP can be captured using explicit $tick$ events, through digitization. In order to tackle the state space explosion problem, we develop a BDD-based symbolic model checking approach to verify Stateful Timed CSP models. Due to the rich language features, BDD-based system encoding and verification is highly nontrivial. In this work, we show how to systematically encode Stateful Timed CSP models in BDD. Our approach consists of two steps. The first step is to identify *maximum primitive components* of a given system and then generate finite state machines (FSMs) from them, applying a set of symbolic firing rules. These FSMs are then encoded in the standard way. The second step is to compose the encoded components using a set of BDD-based compositional functions. The proposed method has been implemented in the PAT model checker. It supports properties like reachability, linear temporal logic, etc. The effectiveness of our technique is evaluated with benchmark systems.

1 Introduction

Real-time systems are a class of systems whose correctness depends on the time at which events occur. Examples of real-time systems ranges from simple timed protocols (like Fischer's protocol) to large complex embedded systems (like signaling systems for high-speed trains). The reactions of these systems must obey all of the timing constraints. In other words, these systems must produce responses not only correctly but also with exact timing. Any violation of these constraints may cause damages and risk the human lives. Therefore it is immediately clear that verification real-time systems is a crucial phase in the design of real-time systems.

* This research is partially supported by TRF project 'Research and Development in the Formal Verification of System Design and Implementation'.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 398–413, 2012.

Timed automata are an extension of finite-state automata equipped with finitely many real-valued clock variables to keep track of time. They are often used to model real-time systems. In timed automata, transitions and states may be labeled with clock constraints. Clock constraints labeled with states, called state invariant, limit the amount of time that may be spent at that state. Clock constraints labeled with transitions, called transition guard, must hold for the transition to be taken. To specify properties, a real-time invariant of CTL, e.g., Timed CTL (TCTL, for short) has been proposed.

Efficient automatic model-checking algorithms for real-time systems have been obtained in recent years. Note that traditional model checking algorithms could not be applied directly to real-time verification because time factors are modeled as continuous and real-value variables. Zone abstraction [4], which groups clock valuations using a convex constraint [4], has emerged as a popular approach and has been employed by tools like UPPAAL. Other approaches have also been proposed, especially for a subset of timed automata, which can be digitized (i.e., closed timed automata [5]). For instance, Lamport [10] argued that model checking of real-time systems can be really simple if digitization is adopted. Digitization translates a real-time verification problem to a discrete one by using clock ticks to represent time elapsing explicitly. The advantage of digitization is that the techniques which are developed for classic automata verification can be applied without the added complexity of zone operations. Though digitization does not preserve the continuous-time semantics of time-automata, it was proved to be sound for a large class of verification problems [5].

Stateful Timed CSP has been recently proposed, as a complementary language to timed automata, to model (and verify) hierarchical real-time systems. It is an expressive modeling language which combines data structure/operations, complicated control flows (modeled using compositional process operators adopted from Timed CSP), and real-time requirements like deadline and within. In [21], it has been show that zone abstraction can be applied to Stateful Timed CSP by dynamically creating/deleting clocks. Unsurprisingly, however, state space explosion remains as a huge challenge. In [19], it has been show that Stateful Timed CSP is equivalent to closed timed automata with silent transitions, which implies that the timing constraints of Stateful Timed CSP can be captured using explicit tick events, through digitization. In this work, inspired by on previous work on combining BDD and digitization [3,13], we develop a BDD-based symbolic model checking approach to verify Stateful Timed CSP. Due to the rich language features, BDD-based system encoding and verification is highly nontrivial.

The contribution of the work is threefold. Firstly, we develop a systematic way of encoding Stateful Timed CSP. A Stateful Timed CSP process can be encoded by two ways: using FSMs and using compositional functions. Primitive components are translated to FSMs based on the Stateful Timed CSP semantics. These FSMs are encoded in BDD and then composed gradually by a rich set of compositional functions. Secondly, we support a range of model checking algorithms. For instance, we are able to verify LTL with the assumption of non-Zenoness. While checking whether or not an execution is zeno is difficult for zone approaches [22,6], in digitization, an execution of a digitized system is non-Zeno if and only if it contains infinitely many clock ticks. Therefore a digitized system is non-Zeno if time advances at least one time unit in all its cycles. In other words, non-Zenoness assumption can be supported by requiring all cycles to contain at least one tick transition. Lastly, we implement our approach in the PAT model checker [20] and evaluate the performance of BDD-based symbolic model checking with zone-based approaches with a number of systems. We show that our approach complements the zone abstraction approach [21] and offers significantly better performance in a number of cases.

Related Work. After the timed automata were introduced in [1], many tools and techniques are proposed, for example, Different Bounded Matrices [4], Clock-Restriction Diagrams [24], and Difference Decision Diagram [12]. Our work was inspired by the digitization which was proposed in [5,10]. However the difference in our symbolic technique is the use of tick transitions to represent explicitly the timing constraints instead of the use of clock variables. Based on this, a BDD encoding library for digitized systems was developed [14]. This paper presents the extension which only focus on verification of Stateful Timed CSP. Our approach is similar to the two-level approach used in FDR [16]. Basically FDR exploits a hybrid high-/low-level approach for calculating the operational semantics of a process. The low level comprises all true recursions while in the high level, processes are composed by parallel composition, hiding and renaming. Identifying low-level processes in FDR is the same as finding the maximum primitive components in our approach. However the ways to tackle the state space explosion in FDR and in our approach are different. In the compiling process on high-level called $super\ compiling$ of FDR, a single LTS is built on-the-fly from other LTSs based on the calculating a set of rules. In contrast in our approach, maximum primitive components are combined by BDD-based compositional functions. Our work in this paper extends the works in [16] because while two-level approach is used to verify un-timed systems, our approach is able to verify real-time systems.

2 Stateful Timed CSP

In this section, we briefly introduce the syntax and the semantics of Stateful Timed CSP processes. The readers are referred to [19] for a complete list of syntax and semantics. Let the label a describe the name of events which are not $tick$ and can be either an external event, a termination event ✓ or an internal event tau, the label c describe channel name and $tick$ denote the passage of one time unit.

A Stateful Timed CSP model is a 3-tuple (Var, σ_0, P_0) where Var is a set of *finite-domain* global variables; σ_0 is the initial valuation of Var (which maps one variable to one value only) and P_0 is a process. A process is a block of computations, which can be defined under Backus-Naur form as Fig. 1.

Process $Stop$ could not make any progress and must still be in the same state after any time period has elapsed. Process $Skip$ is ready to terminate and becomes $Stop$. However some time may elapse before this termination. Process Event Prefixing $a \rightarrow P$ prepares to engage the event a and behaves as P afterward. Similar to $Skip$, delay on this event may occur. Urgent Event Prefixing $a \twoheadrightarrow P$, on the other hand, requires event a to occur as soon as it is enabled. Process Data Operation Prefixing $a\{program\} \rightarrow P$ performs the $program$ with the event a. Note that $program$ can include from simple assignments to complicated sequential structures like if, $while$ and is executed atomically with the

$P = \mathit{Stop} \mid \mathit{Skip}$	– primitives
$\mid\ a \rightarrow P$	– event prefixing
$\mid\ a \twoheadrightarrow P$	– urgent event prefixing
$\mid\ a\{\mathit{program}\} \rightarrow P$	– data operation prefixing
$\mid\ \text{if}(b)\{P\}\ \text{else}\ \{Q\}$	– conditional choice
$\mid\ P \mid Q$	– general choice
$\mid\ P \setminus X$	– hiding
$\mid\ P;\ Q$	– sequential composition
$\mid\ P \parallel Q$	– parallel composition
$\mid\ c?\{\mathit{program}\} \rightarrow P \mid c!\{\mathit{program}\} \rightarrow P$	– Channel Input/Output
$\mid\ Q$	– process referencing
$\mid\ \mathit{Wait}[d]$	– delay*
$\mid\ P\ \mathit{timeout}[d]\ Q$	– timeout*
$\mid\ P\ \mathit{interrupt}[d]\ Q$	– timed interrupt*
$\mid\ P\ \mathit{within}[d]$	– timed responsiveness*
$\mid\ P\ \mathit{deadline}[d]$	– deadline*

Fig. 1. Stateful Timed CSP Process Constructs

event. Process Conditional Choice, defined as $\text{if}(b)\{P\}$ else $\{Q\}$ will behave as P or as Q based on the evaluation of the expression b. Process Unconditional Choice $P \mid Q$ offers an (unconditional) choice between P and Q[1]. Sequential composition $P;\ Q$ behaves as P until P terminates and then behaves as Q immediately. Process $P \setminus X$ hides occurrences of events in X from the environment. In other words, any event in X engaged by P becomes invisible event τ. Parallel composition of two processes P and Q is written as $P \parallel Q$, where P and Q may communicate via event synchronization (following CSP rules [7]) or shared variables. Notice that if P and Q do not communicate through event synchronization, then it is written as $P|||Q$, which reads as 'P interleave Q'. In addition to multi-party synchronization based on event names, Stateful Timed CSP also provides pairwise synchronization via channel communications. Transitions labeled with channel input (or channel output) of a process can not be taken on its own but must be matched by transitions labeled with corresponding channel output (channel input) of another process running in parallel with it. A process may be given a name, written as $P \mathrel{\widehat{=}} Q$, and then referenced through its name. Recursion is allowed by process referencing.

In addition to two traditional timed process constructs Delay (Wait), and Timeout ($\mathit{timeout}$) from Timed CSP, Stateful Timed CSP includes three new process constructs Time Interrupt ($\mathit{interrupt}$), Timed Responsiveness (within) and Deadline ($\mathit{deadline}$). This extension allows us to capture common real-time system behavior patterns easily (all timed process constructs are marked with * in Figure 1). Let $d \in \mathbb{R}_+$. Process $\mathit{Wait}[d]$ idles for exactly d time units before terminating. Process $P\ \mathit{timeout}[d]\ Q$ imposes a constraint on the process P to engage the first visible event within d time units. Otherwise after d time units, process Q takes the execution control. In process $P\ \mathit{interrupt}[d]\ Q$, if P terminates before d time units, $P\ \mathit{interrupt}[d]\ Q$ behaves exactly as P. Otherwise, $P\ \mathit{interrupt}[d]\ Q$ behaves as P until d time units and then Q

[1] For simplicity, we omit external and internal choices [7] in the discussion.

takes over. In contrast to $P\ timeout[d]\ Q$, P may engage in multiple visible events before it is *interrupted*. Process $P\ within[d]$ requires process P to engage an visible event with in d time units. In process $P\ deadline[d]$, P must terminate within d time units, possibly after engaging in multiple visible events. Notice that a timed process construct is always associated with an *integer* constant d which is referred to as its parameter.

Example 1. We use Fischer's mutual exclusion protocol [9] to illustrate system modeling using Stateful Timed CSP. The protocol is designed to guarantee mutually exclusive access to a critical section among competing processes $P(i)$ where $i \in [1..n]$ is the unique identifier of that process. Each process $P(i)$ executes the following algorithm where $lock$ is a shared variable, and initialized with the value 0:

```
repeat
    await($lock = 0$);
    $lock := i$
    delay
until ($lock = i$);
$critical\ section$;
$lock := 0$;
```

Note that **await** $(cond)$ is an abbreviation for **while** $(\neg\ cond)$ **do** $skip$ and **delay** corresponds to an explicit delay statement. The role of the **delay** statement is that it guarantees while it delays itself, other processes after passing the **await** statement must finish the assignment $lock := i$. The correctness of the protocol depends on the assumptions about the time taken to read and write to the shared variable $lock$, and the delay length. It was shown that the mutual exclusion is guaranteed if the upper bound a on the time taken at the assignment $lock := i$ is less than the lower bound b on the delay length. Because other reading and writing statements to the shared variable $lock$ is not important, we will not impose any timing constraint on them. The protocol can be modeled as a Stateful Timed CSP model $(Var, \sigma_0, Fischer)$ where $Var = \{lock\}$ and $\sigma_0(lock) = 0$ and process $Fischer$ is defined as: $P(1) ||| \cdots ||| P(n)$ where

```
$P(i)$        $\widehat{=}$ $if\,(lock = 0)\{$
                     $(setLock\{lock := i\} \rightarrow Skip)\ deadline[a]$;
                     $Wait[b]$;
                     $if\,(lock = i)\{$
                         $Critical(i)$
                     $\}else\{$
                         $P(i)$
                     $\}$
                 $\}$;
$Critical(i)$ $\widehat{=}$ $enter \rightarrow exit\{lock := 0\} \rightarrow P(i)$;
```

Process $Fischer$ is the Interleave composition of $P(1) ||| \cdots ||| P(n)$. Each process $P(i)$ has an unique identifier described as i. As we can see in the model, timing constraints on each operation can be translated straightforwardly using the set of timed process constructs. For example, $(setLock\{lock := i\} \rightarrow Skip)\ deadline[a]$ imposes a constraint on the event $setLock$, i.e., it must occur within a time units. The **delay** statement which delays at least b time units can be expressed as $Wait[b]$. Note that after waiting exactly b time units in $Wait[b]$, the process $P(i)$ behaves as the process if$(lock = i)\{\cdots\}$. Since we do not put any constraint on this process, it can idle as long as it wants. Therefore in total the process $P(i)$ can delay at least b time units before entering the critical section. □

There are two approaches to verify Stateful Timed CSP. One is based zone abstraction, which has been proposed in [19]. The other is through digitization, since it has been proved that Stateful Timed CSP is equivalent to some variant of closed timed automata [19]. On one hand, while zone abstract works well in many examples, its complexity is exponential in the number of clocks and its performance in practice can be strongly related to ratio of constants appearing in the clock constraints. For instance, in the Leader Algorithm (which has a very small maximal constants of clock constraints), Uppaal's execution time is strongly dependent on the ratio $MsgDelay/Period$ [10]. Specifically for ratios greater than 0.6, Uppaal easily runs out of memory. On the other hand, though digitization suffers from large clock upper bounds (which imply a large number of tick events), it is not affected by the ratio of the constants. Furthermore, some problems like the non-Zenoness checking problem are much easier with digitization. We thus proposed an approach complementary to the zone abstraction approach in [19], using BDD and digitization to verify Stateful Timed CSP.

3 BDD Encoding

In this section, we show how we systematically encode Stateful Timed CSP processes in BDD. There are two ways. One is to generate an FSM for each Stateful Timed CSP process and encode the FSM in the standard way. The other is to define a set of BDD compositional functions according to the process construct semantics and then encode Stateful Timed CSP processes into BDDs directly without the FSM construction. Both have their own advantages and therefore are used in different cases.

We remark that Stateful Timed CSP is expressive enough so that a process expression generated by the operational semantics may be unbounded. For example, define $P_0 = e \rightarrow (P_0 ||| P_{new})$ which forks a process P_{new} every time e occurs. The resultant process therefore may contain unboundedly many copies of P_{new}. In this work, we assume that a process always has a bounded length, following [17,15].

3.1 Encoding Stateful Timed CSP Processes with FSMs

An FSM is a tuple $\mathcal{M} = (Var, S, init, Act, T)$ such that Var is a set of finite-domain variables; S is a finite set of control states; $init \in S$ is the initial state; Act is the alphabet of events and channels; and T is a labeled transition relation. A transition label is of the form $[guard]evt\{prog\}$ where $guard$ is an optional guard condition constituted by variables in Var; evt is either an event name, a channel input/output or the special $tick$ event (which denotes 1-unit elapsed time); and $prog$ is an optional transaction, i.e., a sequential program which updates global/local variables. A transaction (which may

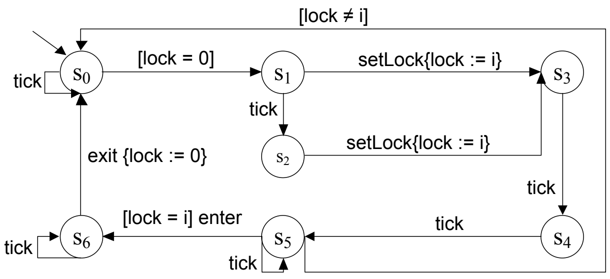

Fig. 2. The FSM of Process $P(i)$

contain program constructs like *while*-*do*) associated with a transition is to be executed atomically. A non-atomic operation can be broken into multiple transitions. A transition is possible if the $guard$ is true given current valuation σ of Var. Moreover a transition labeled with channel input/output can not occur by itself but must be synchronized with the transition labeled with corresponding channel output/input.

The operational semantics of Stateful Time CSP allows us to interpret Stateful Time CSP processes as FSMs. For example, we can manually draw the FSM shown in Fig. 2 for the process $P(i)$ of Fischer's protocol in the Example 1 with $a = 1$ and $b = 2$. However translating from a Stateful Time CSP process to an FSM in general is not trivial. In this following, we show how to systematically build the corresponding FSM from a Stateful Timed CSP process. This approach relies on symbolic firing rules, which are different from concrete firing rules in [21] as variables valuations are irrelevant. Specifically the symbolic firing rules are used to generate the whole control flow of a certain process. In other words, the valuation of variables and the effect of transactions are ignored at this step, but they will be considered when transactions are encoded in BDD. For instance, the symbolic firing rule of process Data Operation Prefixing $Q = b\{x := x + 1\} \rightarrow R$ says that at the process Q, if the transition labeled with $b\{x := x + 1\}$ is taken, it will behave as R. In contrast concrete firing rules say, e.g., at the process Q, suppose the current value of x is 0, then after the transition is taken, it will be have as R and the value of x becomes 1. The concrete firing rules, therefore, are used to generate on-the-fly the whole state space explicitly. So different uses of firing rule are suitable for different purposes. In this work, symbolic firing rules are used to generate the corresponding FSM systematically and effectively. Our symbolic firing rules follow the form in [18]:

$$\frac{\begin{array}{l} \mathit{antecedent}\ 1 \\ \ldots \\ \mathit{antecedent}\ n \end{array}}{\mathit{conclusion}}\ [\ \mathit{side\ condition}\ ]$$

The conclusion can be deduced if all the antecedents are true and the side condition is also true. In the case where antecedents or side condition are missing, they are considered

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P} \qquad \frac{}{(a \rightarrow P) \xrightarrow{tick} (a \rightarrow P)}$$

$$\frac{P_0 \xrightarrow{[g]a\{p\}} P_0'}{\begin{array}{c} P_0|||P_1 \xrightarrow{[g]a\{p\}} P_0'|||P_1 \\ P_1|||P_0 \xrightarrow{[g]a\{p\}} P_1|||P_0' \end{array}} \; [\, a \neq \checkmark \,] \qquad \frac{\begin{array}{c} P_0 \xrightarrow{tick} P_0' \\ P_1 \xrightarrow{tick} P_1' \end{array}}{P_0|||P_1 \xrightarrow{tick} P_0'|||P_1'}$$

$$\frac{\begin{array}{c} P_0 \xrightarrow{[g_0]\checkmark\{p_0\}} P_0' \\ P_1 \xrightarrow{[g_1]\checkmark\{p_1\}} P_1' \end{array}}{P_0|||P_1 \xrightarrow{[g_0 \wedge g_1]\checkmark\{p_0;\, p_1\}} P_0'|||P_1'} \qquad \frac{\begin{array}{c} P_0 \xrightarrow{[g_0]c?\{p_0\}} P_0' \\ P_1 \xrightarrow{[g_1]c!\{p_1\}} P_1' \end{array}}{\begin{array}{c} P_0|||P_1 \xrightarrow{[g_0 \wedge g_1]c\{p_0;\, p_1\}} P_0'|||P_1' \\ P_1|||P_0 \xrightarrow{[g_0 \wedge g_1]c\{p_0;\, p_1\}} P_1'|||P_0' \end{array}}$$

$$\frac{}{Wait[t] \xrightarrow{tick} Wait[t-1]} \; [\, t \geq 1 \,] \qquad \frac{}{Wait[0] \xrightarrow{\tau} SKIP}$$

$$\frac{P_0 \xrightarrow{a} P_0'}{P_0 \; within[t] \xrightarrow{a} P_0'} \qquad \frac{P_0 \xrightarrow{\tau} P_0'}{P_0 \; within[t] \xrightarrow{\tau} P_0' \; within[t]}$$

$$\frac{P_0 \xrightarrow{tick} P_0'}{P_0 \; within[t] \xrightarrow{tick} P_0' \; within[t-1]} \; [\, t \geq 1 \,]$$

Fig. 3. Sample Symbolic Firing Rules

as vacuously true. A number of conclusions which can be drawn from the same set of antecedents and side condition can be grouped below the line one after the other.

The FSM generation procedure basically works as follow. Each process P is mapped with a state in the FSM called 'state P' and this state is also the initial state of that process's FSM. There is a transition labeled with $[guard]\, evt\{prog\}$ from state P to state P' when the relation $P \xrightarrow{[guard]\, evt\{prog\}} P'$ can be deduced from the rules. The symbolic firing rules are applied until there is no new state generated. In the following, we present the sample symbolic firing rules of Event Prefixing, Interleave, Delay, and Timed Responsiveness process constructs.

- Given any process Event Prefixing $a \rightarrow P$, there is a transition labeled with event a from the state $a \rightarrow P$ to the state P. In addition, there is a transition label with event $tick$ looping at the state $a \rightarrow P$. For the events marked as urgent, this looping transition labeled with event $tick$ is not available. It forces the process to engage the event without any delay.

- Based on rules of process Interleave, all of the subprocesses in the Interleave composition must synchronize with the termination ✓ and $tick$ events. Moreover channel in transition labeled with $c?$ from one process can be combined with channel out transition labeled with $c!$ from another process to be promoted as c. When transitions are synchronized, we constraint transactions of these transition are not conflict and the execution order of transactions are not important. Other events occur interleave. In addition in the symbolic firing rules of this process and also of other processes, $tick$ transitions are never attached with any guard condition and any transaction. They are simple as a direct sequence of the use $tick$ transitions to explicitly represent the timing constraints. This simplicity helps us to have more optimal BDD encoding of the $tick$ transitions.
- $Tick$ transitions are used to track the passage of one time unit in the symbolic firing rules of process Delay $Wait[t]$. Specifically there is a transition labeled with $tick$ from the state $Wait[t]$ to state $Wait[t-1]$. After delaying itself, it will behave as $SKIP$ by the τ transition from state $Wait[0]$ to state $SKIP$.
- The last three rules are the symbolic firing rules of process construct Timed Responsiveness $P_0\ within[t]$. These rules are self-explanatory. $Tick$ transitions are used to track the passage of time. Unless a visible event is engaged, the timed responsiveness condition is not resolved.

Example 2. Process $P(i)$ of Fischer's protocol in the Example 1 is used again as illustration. However for simplicity all the states are renamed to $s_0, \cdots, s_6$ and we will explain the FSM generation procedure starting at process $Critical(i)$ whose corresponding state is the state s_5. According to the firing rules of process Event Prefixing in Fig. 3, in the FSM of the process $P(i)$, there is a transition labeled with $[lock = i]enter$ from the state $Critical(i)$ (state s_5) to state $exit\{lock := i\} \rightarrow P(i)$ (state s_6), and a transition labeled with $tick$ looping at the state of process $Critical(i)$ (state s_5). Then by applying those firing rules again for the process $exit\{lock := i\} \rightarrow P(i)$, there is another transition labeled with $exit\{lock := i\}$ from the state $exit\{lock := i\} \rightarrow P(i)$ (state s_6) back to the state $P(i)$ (state s_0), and a transition labeled with $tick$ looping at the state $exit\{lock := i\} \rightarrow P(i)$ (state s_6). The FSM generation procedure is stopped because there is no new state created. □

Before giving explanation how to encode an FSM, we will briefly describe how to encode a finite set. Essentially given any finite set X, encoding X is to enumerate elements of X in binary and represent them as Boolean functions. Therefore to encode X, we need n boolean variables $x_0, \cdots, x_{n-1}$ where $n = \lceil log_2 |X| \rceil$. Then each element in X is mapped with a bit vector $(x_0, \cdots, x_{n-1})$ by an injective encoding function $f_X : X \rightarrow \{0,1\}^n$. Note that this mapping is fixed throughout the BDD encoding. For instance, encoding the set of four elements $X = \{a, b, c, d\}$ requires two boolean variables x_0 and x_1. The encoding functions f_X is defined as $f_X(a) = (0,0)$, $f_X(b) = (0,1)$, $f_X(c) = (1,0)$, and $f_X(d) = (1,1)$. As a result the predicate of the subset $Y = \{a, b\}$ is $((x_0, x_1) = f_X(a) \vee (x_0, x_1) = f_X(b))$. For simplicity we will use the label x to denote the bit vector $(x_0, \cdots, x_{n-1})$. Therefore the predicate of the subset Y can be rewritten shortly as $(x = f_X(a) \vee x = f_X(b))$. Using this technique, we can encode the set of states and the set of event names and channel names in an FSM. Moreover we can also encode all the data types whose domain is finite, e.g., boolean, integer, array of booleans, and array of integers. To encode transitions, each variable x in $\overrightarrow{V} \cup \overrightarrow{v}$ has another copy called x' which denotes the variable x's value after the transition.

The BDD encoding of an FSM, referred to as a *BDD machine*, is a tuple $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, \mathit{Init}, \mathit{Trans}, \mathit{Out}, \mathit{In}, \mathit{Tick})$. $\overrightarrow{V}$ is a set of unprimed Boolean variables encoding global variables, event names and channel names, which are fixed for the whole system before encoding. $\overrightarrow{v}$ is a set of variables encoding local variables and local control states; Init is a formula over $\overrightarrow{V}$ and $\overrightarrow{v}$ encoding the initial valuation of the variables. Trans is the encoding of transitions *excluding synchronous channel input/output and tick-transitions*. Out (In) is the encoding of synchronous channel output (input). Note that transitions in Out and In are to be matched by corresponding transitions in In and Out respectively from the environment and are thus separated from the rest of the transitions. Tick is also the encoding of transitions labeled with tick. Then the final transition function of an FSM is taken from Trans and Tick. In other words, it can engage an action or idle one time unit. We still calculate Out and In and separate them from Trans and Tick because transitions from Out and In can be useful if they are synchronized.

Let *BDD machine* $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, \mathit{Init}, \mathit{Trans}, \mathit{Out}, \mathit{In}, \mathit{Tick})$ be the encoding of an FSM $\mathcal{M} = (\mathit{Var}, S, \mathit{init}, \mathit{Act}, T)$ where

- $\overrightarrow{V} = V_1 \cup \mathit{Events}$ where V_1 and $\mathit{Events} = \{\mathit{event}_0, \cdots, \mathit{event}_{n-1}\}$ are the sets of boolean variables to encode global variables and the alphabet Act respectively. Let event denote the bit vector $(\mathit{event}_0, \cdots, \mathit{event}_{n-1})$.
- $\overrightarrow{v} = v_1 \cup \mathit{States}$ where v_1 and $\mathit{States} = \{\mathit{state}_0, \cdots, \mathit{state}_{m-1}\}$ are the sets of boolean variables to encode local variables and the set of states S respectively. Similarly let state denote the bit vector $(\mathit{state}_0, \cdots, \mathit{state}_{m-1})$. Moreover for any global or local variable x, let the same label x denote the corresponding bit vector of boolean variables to encode that variable. Note that these labels x are different. The former x is the variable declared in the model while the latter x is a shorthand for a bit vector in the BDD encoding functions.
- $\mathit{Init} = (\mathit{state} = f_S(\mathit{init}))$
- $\mathit{Trans} = \bigvee(\mathit{state} = f_S(s_0) \wedge g_{bdd} \wedge \mathit{event}' = f_{Act}(e) \wedge \mathit{prog}_{bdd} \wedge \mathit{state}' = f_S(s_1))$ for all transitions from state s_0 to state s_1 labeled with $[g]e\{\mathit{prog}\}$ (where $e \neq \mathit{tick}$). For simplicity, we skip how we encode guard expression g to g_{bdd} and program block prog to prog_{bdd}. Interested readers can refer to [13].
- $\mathit{Out} = \bigvee(\mathit{state} = f_S(s_0) \wedge g_{bdd} \wedge \mathit{event}' = f_{Act}(e) \wedge \mathit{prog}_{bdd} \wedge \mathit{state}' = f_S(s_1))$ for all transitions from state s_0 to state s_1 labeled with a synchronous channel output e, guarded with g and attached with transaction prog.
- $\mathit{In} = \bigvee(\mathit{state} = f_S(s_0) \wedge g_{bdd} \wedge \mathit{event}' = f_{Act}(e) \wedge \mathit{prog}_{bdd} \wedge \mathit{state}' = f_S(s_1))$ for all transitions from state s_0 to state s_1 labeled with a synchronous channel input e, guarded with g and attached with transaction prog.
- $\mathit{Tick} = \bigvee(\mathit{state} = f_S(s_0) \wedge \mathit{event}' = f_{Act}(\mathit{tick}) \wedge \mathit{state}' = f_S(s_1))$ for all tick transitions from state s_0 to state s_1.

Example 3. The *BDD machine* $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, \mathit{Init}, \mathit{Trans}, \mathit{Out}, \mathit{In}, \mathit{Tick})$ of the FSM in Fig. 2 is as follow:

- $\overrightarrow{V} = \mathit{Lock} \cup \{\mathit{event}_0, \mathit{event}_1\}$ where Lock is the set of boolean variables to encode the shared variable lock.
- $\overrightarrow{v} = \{\mathit{state}_0, \mathit{state}_1, \mathit{state}_2\}$. Note that the process parameter i in the definition of $P(i)$ is constant and is replaced with its value before the encoding. In the below encoding functions of Trans and Tick, we still keep i to show generally how all processes $P(i)$ in the Fischers' protocol are encoded.
- $\mathit{Init} = (\mathit{state} = f_S(s_0))$
- $\mathit{Trans} = (\mathit{state} = f_S(s_0) \wedge \mathit{lock} = 0 \wedge \mathit{state}' = f_S(s_1))$
  $\vee\ (\mathit{state} = f_S(s_1) \wedge \mathit{event}' = f_{Act}(\mathit{setLock}) \wedge \mathit{lock}' = i \wedge \mathit{state}' = f_S(s_3))$
  $\vee\ (\mathit{state} = f_S(s_2) \wedge \mathit{event}' = f_{Act}(\mathit{setLock}) \wedge \mathit{lock}' = i \wedge \mathit{state}' = f_S(s_3))$
  $\vee\ (\mathit{state} = f_S(s_5) \wedge \mathit{lock} \neq i \wedge \mathit{state}' = f_S(s_0))$
  $\vee\ (\mathit{state} = f_S(s_5) \wedge \mathit{lock} = i \wedge \mathit{event}' = f_{Act}(\mathit{enter}) \wedge \mathit{state}' = f_S(s_6))$
  $\vee\ (\mathit{state} = f_S(s_6) \wedge \mathit{event}' = f_{Act}(\mathit{exit}) \wedge \mathit{lock}' = 0 \wedge \mathit{state}' = f_S(s_0))$
- $\mathit{Out} = \mathit{In} = \mathit{false}$
- $\mathit{Tick} = (\mathit{state} = f_S(s_0) \wedge \mathit{event}' = f_{Act}(\mathit{tick}) \wedge \mathit{state}' = f_S(s_0))$
  $\vee\ (\mathit{state} = f_S(s_1) \wedge \mathit{event}' = f_{Act}(\mathit{tick}) \wedge \mathit{state}' = f_S(s_2))$
  $\vee\ (\mathit{state} = f_S(s_3) \wedge \mathit{event}' = f_{Act}(\mathit{tick}) \wedge \mathit{state}' = f_S(s_4))$
  $\vee\ (\mathit{state} = f_S(s_4) \wedge \mathit{event}' = f_{Act}(\mathit{tick}) \wedge \mathit{state}' = f_S(s_5))$
  $\vee\ (\mathit{state} = f_S(s_5) \wedge \mathit{event}' = f_{Act}(\mathit{tick}) \wedge \mathit{state}' = f_S(s_5))$
  $\vee\ (\mathit{state} = f_S(s_6) \wedge \mathit{event}' = f_{Act}(\mathit{tick}) \wedge \mathit{state}' = f_S(s_6))$ □

3.2 Encoding Stateful Timed CSP Processes with Compositional Functions

By using the approach presented in the last section, in theory we can translate any Stateful Timed CSP process to an FSM and encoding it. However we do not apply that approach to generate the FSM of parallel processes. Because in the FSM of a parallel composition, the numbers of states and transitions grow exponentially with the number of subprocesses running in parallel. Especially it becomes completely redundant when guards and transactions of the transitions in a certain sub-process are encoded to BDD many times. For example, if we apply the FSM generation procedure to the process $P_1 ||| P_2$, suppose the state of that FSM is of the form (s_1, s_2) where s_1, and s_2 are states in the FSMs of P_1 and P_2 respectively. For any transition t from state s_1 to s_1' in the FSM of P_1, there is a corresponding transition from state (s_1, s_2) to state (s_1', s_2) in the FSM of $P_1 ||| P_2$. Obviously the guard and the transaction of the transition t will be encoded m times where m is the number of states in the FSM of P_2. These overheads make encoding of parallel processes with FSMs inefficient. Therefore we provide compositional functions to encode parallel processes without translating it to FSMs. As a result, compositional functions for all kinds of processes are required to be provided because after using the compositional function, the FSM is no longer available and only compositional functions can be used.

In the following, we will show how to encode two kinds of Stateful Timed CSP processes: Interleave and Timed Responsiveness processes with compositional functions. We fix two BDD machines $\mathcal{B}_i = (\overrightarrow{V}, \overrightarrow{v}_i, \mathit{Init}_i, \mathit{Trans}_i, \mathit{Out}_i, \mathit{In}_i, \mathit{Tick}_i)$, $i \in \{0, 1\}$, which are the encoding of processes P_i. $\overrightarrow{v}_0$ and $\overrightarrow{v}_1$ are disjoint and $\overrightarrow{V}$ is always shared. Symbolic firing rules of Interleave and Timed Responsiveness process constructs in Fig. 3 can be refered to follow the compositional encoding. Interested readers can refer to [13] for the complete list.

Interleave: Process Interleave can contains 2 or more subprocesses running in parallel. Different from process Parallel these processes are only synchronized in termination event ✓ (still has pairwise synchronization in channel communication like Parallel). Let $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, \mathit{Init}, \mathit{Trans}, \mathit{Out}, \mathit{In}, \mathit{Tick})$ be the BDD machine encoding of the Interleave composition of two processes P_0 and P_1 such that:

- $\overrightarrow{v} = \overrightarrow{v}_0 \cup \overrightarrow{v}_1$;
- $\mathit{Init} = \mathit{Init}_0 \wedge \mathit{Init}_1$.
- $\mathit{Trans} = \bigvee_{i \in \{0,1\}}[(\mathit{Trans}_i \wedge \mathit{event}' \neq f_{Act}(\checkmark) \wedge \overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i}) \vee (\mathit{In}_i \wedge \mathit{Out}_{1-i}) \vee (\mathit{Trans}_i \wedge \mathit{Trans}_{1-i} \wedge \mathit{event}' = f_{Act}(\checkmark))]$. Trans includes 3 kinds of transitions: local transitions from each component, synchronous channel communication and synchronous termination transition. $(\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$ denotes that the local variables of $\mathcal{B}_{1-i}$ are unchanged.
- $\mathit{In} = \bigvee_{i \in \{0,1\}}(\mathit{In}_i \wedge \overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$
- $\mathit{Out} = \bigvee_{i \in \{0,1\}}(\mathit{Out}_i \wedge \overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$
- $\mathit{Tick} = \mathit{Tick}_0 \wedge \mathit{Tick}_1$

Within: In $P_0\ \mathit{within}[t]$ process, process P_0 is forced to engage a visible event within the given t time units. Let $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, \mathit{Init}, \mathit{Trans}, \mathit{Out}, \mathit{In}, \mathit{Tick})$ be the BDD machine encoding of $P_0\ \mathit{within}[t]$ where

- $\overrightarrow{v} = \overrightarrow{v}_0 \cup \{\mathit{clk}\}$, $-1 \leq \mathit{clk} \leq t$ records the number of elapsed time units so far and $\mathit{clk} = -1$ indicates an visible action is engaged.
- $\mathit{Init} = (\mathit{Init}_0 \wedge \mathit{clk} = 0)$
- $\mathit{Trans} = \mathit{clk} \leq t \wedge \mathit{Trans}_0 \wedge [(\mathit{event} \neq f_{Act}(\tau) \wedge \mathit{clk}' = -1) \vee (\mathit{event}' = f_{Act}(\tau) \wedge \mathit{clk}' = \mathit{clk})]$
- $\mathit{In} = \mathit{clk} < t \wedge \mathit{In}_0 \wedge \mathit{clk}' = -1$
- $\mathit{Out} = \mathit{clk} < t \wedge \mathit{Out}_0 \wedge \mathit{clk}' = -1$
- $\mathit{Tick} = \mathit{Tick}_0 \wedge [(\mathit{clk} \geq 0 \wedge \mathit{clk} < t \wedge \mathit{clk}' = \mathit{clk} + 1) \vee (\mathit{clk} = -1 \wedge \mathit{clk}' = -1)]$

Note that a channel communication is clearly a visible event. Thus if channel communication occurs, variable clk is assigned -1 to mark the happening of that visible event.

As we can observe, except parallel processes, encoding processes with compositional functions is not as optimal as with FSMs. Unlike encoding with FSMs, many auxiliary variables are introduced in the encoding with compositional functions to control the flow, for example, clk variable in Timed Responsiveness to record the number of elapsed time units. Therefore our strategy for encoding a Stateful Timed CSP process is to find its maximum primitive components which can be translated to FSMs and then encode these FSMs as BDD machines. Identifying the maximum primitive components is straightforward because maximum primitive components are the maximum components whose definitions do not contain Parallel/Interleave process construct. Finally these BDD machines are composed to achieve the final BDD machine of the given process. For instance, in Example 1, the identified maximum primitive components are

n processes $P(i)$ where $i \in \{1, \cdots, n\}$. Next, FSMs translated from these subprocesses are encoded as BDD machines, which are then composed using the Interleave compositional function to generate the BDD encoding of the process $Fischer$.

3.3 Limitations on BDD Encoding

Stateful Timed CSP is too expressive to be fully encoded. Consequently there are some Stateful Timed CSP processes which are not possible to be encoded. Firstly processes having varying parameters are not supported. An example of processes having a varying parameter is $P(i) = a \rightarrow P(i+1)$. The reason of this limitation is because of the update of the parameter $i := i + 1$ when the process starts to behave as P again. This update must be done somewhere before the process behaves as P again. There are two possible ways to deal with this, one is to attach the parameter updates on the immediately precedent event (in this example it is the event a), another is to create a separate transition to update the process parameters. However both ways have problems which may change the semantics of the defined process. In the first way, these parameter updates could conflict with each other at the precedent event. An illustration of this problem is $Q = a \rightarrow (P(1) \mid P(2))$ where after event a, there is a choice between $P(1)$ and $P(2)$. Therefore we have two conflict updates of the process parameter i of process P, $i := 1$ and $i := 2$. In the second way, by introducing new transitions which updates process parameters, there is a question on the semantics of these transitions, specifically whether these transitions can resolve the choice. If these transitions do not resolve the choice, in the last example, two transitions which update $i := 1$ and $i := 2$ respectively can happen before the choice is resolved. This is similar to the problem in the first way where there are conflicts between these parameter updates. On the other hand, if these transitions can resolve the choice, suppose that in the last example, $P(1)$ could not engage any event while $P(2)$ can, then the process can take the transition which updates $i := 1$ and resolve the choice in favor of the process $P(1)$. After that the process becomes deadlock. However this could not happen because since $P(1)$ is deadlock, the choice must be resolved in favor of $P(2)$.

Secondly encoding with compositional functions could not be applied to recursive processes, e.g., $P = a \rightarrow P$. Based on the Stateful Timed CSP semantics, encoding with compositional function is used to achieve the encoding of a process based on the known encodings of subprocesses. Therefore it is obvious that using compositional functions on a process whose definition has a reference call to itself is not possible and will create an infinite recursive calls of the compositional functions.

In summary there are two restrictions on BDD encoding of Stateful Timed CSP. One restriction is that processes must have constant parameters. However there is a small number of models requiring varying parameters. Moreover global variables can be used to alleviate the restriction. By promoting each varying process parameter with a corresponding global variable and manually attaching the update of those global variables to the suitable events, an equivalent model can be achieved. The other restriction is that compositional encoding is not available for recursive processes and yet this restriction is inevitable. Remember that introducing compositional functions is to optimize the encoding of the parallel process which is the main cause of the state space explosion. After the use of compositional functions, only encoding by compositional functions is possible. However in our experience often the recursive processes do not contain parallel composition. Consequently these processes can be encoded using FSMs.

4 Implementation and Evaluation

Our technique has been implemented as part of the PAT framework [20]. It is based on the CUDD package, with about thirty classes and thousands of lines of C# code. The implementation includes two parts: encoding and verification. The encoding part has functions to generates the FSM from Stateful Timed CSP processes. The advantage of our technique is that the FSM generation procedure is very simple, yet systematical and efficient. For each process construct we only need to define what transitions can be taken from that process and then these transitions are added from the state of the current process. This procedure is called recursively in subsequent processes. In addition to the FSM generation procedure, the encoding part also contains a function to encode an FSM and a set of compositional functions for all process constructs. The second part is the verification which supports a range of properties, e.g., reachability and deadlock analysis or LTL. Verification of LTL is based on a symbolic implementation of the automata-based approach [8,23]. By using digitization technique, verification of real time system, specifically Stateful Timed CSP becomes feasible. Digitization translates a real-time verification problem to a discrete one by using clock ticks to represent elapsed time. Therefore the current model checking algorithm for concurrent systems can be applied without the added complexity of zone operations. Moreover verification of LTL with non-Zenoness assumption can also be supported by converting the non-Zenoness assumptions as $justices$ conditions (weak fairness) [8]. In the following, we evaluate our technique in verification Stateful Timed CSP by comparing its performance with the zone-based approach in PAT in many examples. All models are available online [13]. The test bed is a PC with Intel Core 2 Duo E6550 CPU at 2.33GHz and 3GB RAM.

According to the experiment results in Table 1, in the verification of three mutual exclusion protocols Fischer's protocol [9], AT92 [2], and LTS92 [11], BDD-based approach consistently outperforms Zone-based approach. BDD-based approach is not only faster but also uses less memory than Zone-based approach. For instance, in Fischer protocol of 6 processes, zone-based approach takes more than 1000 seconds and 215 MBs while BDD-based takes only 6 seconds and 101 MBs. Moreover zone-based approach runs out of memory with Fischer protocol of 7 processes, yet BDD-based approach can verify the protocol of up to 12 processes. However in the verification of Train Controller, zone-based approach is much better than BDD-based approach. For instance, in the Railway Controller with 6 trains, zone-based approach only takes 4 seconds and 18 MBs but BDD-based approach takes 905 seconds and 1458 MBs. The reason for this considerable change in performance of BDD-based approach is because of two issues. First the size of BDDs is very sensitive to large clock values. In this benchmark, we set the maximal clock constant to small values, e.g., 4 for Fischer's protocol and 3 for others. Second the size of BDDs is also very sensitive with the FSMs of processes. After examining many examples, we find that there are some models where it is difficult to fully take advantage of the data-sharing capability of BDDs. This is the reason why although we have reduced the maximal clock constants to a very small

Table 1. Compare Zone-based Approach and BDD-based Approach

Model	#Processes	Zone		BDD	
		Time (s)	Memory (MB)	Time (s)	Memory (MB)
Fischer	5	44	19	2	43
Fischer	6	1283	215	6	101
Fischer	7	x	x	17	231
Fischer	12	x	x	1112	1353
AT92	3	7	26	1	22
AT92	4	770	524	2	36
AT92	5	x	x	14	163
AT92	8	x	x	2880	1684
LS92	4	2	13	1	24
LS92	5	1292	76	1	35
LS92	6	x	x	3	57
LS92	15	x	x	996	1406
Railway Controller	5	1	10	51	650
Railway Controller	6	4	18	905	1458
Railway Controller	7	24	18	x	x
Railway Controller	8	201	557	x	x

value, the BDD-based approach's performance is still much poorer than zone-based approach in the Railway Controller example. In contrast if data-sharing occurs a lot in BDDs, the efficiencies of BDD would be higher. This can be shown when we increase the maximal clock constants of the first three protocols up to 20, BDD-based approach still outperforms zone-based approach. Specifically BDD-based approach can verify Fischer's protocol of 10 processes, AT92 of 5 processes and LTS92 of 8 processes. In summary there are some models where zone-based approach performs well while there are other models where BDD-based approach performs well. This experiment shows that these two approaches complements each other.

5 Conclusion

We have illustrated our approach to verify Stateful Timed CSP by using BDD and digitization. We have also presented how Stateful Timed CSP processes are systematically encoded with FSMs and compositional functions. Furthermore our experiments show that there is no superior approach but these approaches have different but complementary advantage.

References

1. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126, 183–235 (1994)
2. Alur, R., Taubenfeld, G.: Results about Fast Mutual Exclusion. In: IEEE Real-Time Systems Symposium, pp. 12–22 (1992)

3. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: A Tool for BDD-Based Verification of Real-Time Systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 122–125. Springer, Heidelberg (2003)
4. Dill, D.L.: Timing Assumptions and Verification of Finite-State Concurrent Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
5. Henzinger, T.A., Manna, Z., Pnueli, A.: What Good Are Digital Clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
6. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Efficient Emptiness Check for Timed Büchi Automata. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 148–161. Springer, Heidelberg (2010)
7. Hoare, C.A.R.: Communicating Sequential Processes. International Series in Computer Science. Prentice-Hall (1985)
8. Kesten, Y., Pnueli, A., Raviv, L.-O.: Algorithmic Verification of Linear Temporal Logic Specifications. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 1–16. Springer, Heidelberg (1998)
9. Lamport, L.: A Fast Mutual Exclusion Algorithm. ACM Trans. Comput. Syst. 5(1), 1–11 (1987)
10. Lamport, L.: Real-Time Model Checking Is Really Simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
11. Lynch, N.A., Shavit, N.: Timing-Based Mutual Exclusion. In: IEEE Real-Time Systems Symposium, pp. 2–11 (1992)
12. Møller, J.B., Hulgaard, H., Andersen, H.R.: Symbolic Model Checking of Timed Guarded Commands Using Difference Decision Diagrams. J. Log. Algebr. Program. 52-53, 53–77 (2002)
13. Nguyen, T.K., Sun, J., Liu, Y., Dong, J.S., Liu, Y.: BDD-based Discrete Analysis of Timed Systems (2012), http://www.comp.nus.edu.sg/%7Epat/bddlib
14. Nguyen, T.K., Sun, J., Liu, Y., Dong, J.S., Liu, Y.: Improved BDD-Based Discrete Analysis of Timed Systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 326–340. Springer, Heidelberg (2012)
15. Ouaknine, J., Worrell, J.: Timed CSP = Closed Timed Safety Automata. Electrical Notes Theoretical Computer Science 68(2) (2002)
16. Palikareva, H., Ouaknine, J., Roscoe, B.: Faster FDR Counterexample Generation Using SAT-Solving. ECEASST 23 (2009)
17. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, Heidelberg (1995)
18. Schneider, S.: Concurrent and Real-Time Systems: The CSP Approach. Wiley (2000)
19. Sun, J., Liu, Y., Dong, J.S., Liu, Y., Shi, L., André, E.: Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP. TOSEM (to appear, 2012)
20. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
21. Sun, J., Liu, Y., Dong, J.S., Zhang, X.: Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 581–600. Springer, Heidelberg (2009)
22. Tripakis, S.: Verifying Progress in Timed Systems. In: Katoen, J.-P. (ed.) ARTS 1999. LNCS, vol. 1601, pp. 299–314. Springer, Heidelberg (1999)
23. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS, pp. 332–344. IEEE Computer Society (1986)
24. Wang, F.: Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram. In: FORTE, pp. 235–250 (2001)